Technical Disclosure Commons

Defensive Publications Series

April 01, 2019

Code generation from language-compliant templates

Christopher J. Phoenix

Code generation from language-compliant templates

ABSTRACT

Programs that auto-generate code generally produce output from a combination of fixed text and variable text. The variable text comes from an input file; the fixed text is typically scattered throughout the program code. This disclosure presents techniques that extract the fixed text from a schema or template file written in the target language, thereby making is easier to auto-generate accurate code.

KEYWORDS

- auto-generated code
- code generation
- template
- schema
- compiler

BACKGROUND

A recurring theme in software engineering is the automatic generation of source code in a variety of formats, e.g., java, C++ source, C++ header, protobuf, etc. For example, output can be auto-generated from a combination of fixed and variable text, where the variable text comes from an input file, and the fixed text is (typically) scattered throughout the program code of a code generator, which may contain a different set of program code for each output language or format. Such auto-generated code finds use, e.g., in creating HTML code, in interfaces between programs in a system, between systems or between programming languages, etc. Examples of code-generation tools which require substantial code to be written include *yacc*, *swig*, *clang*, etc.

DESCRIPTION

There are certain advantages if the input to a general-purpose code-generating program includes a template similar to the desired output file:

- The template file can be adapted from actual software that had been written prior to, or without specialized knowledge of, the use of the text as a template. In other words, ordinary source code can be adapted as input.
- It may be easier for a programmer to read the template file, understand it, and verify its correctness.
- With possibly minor modifications, the template file can be compiled and run, and thus tested for syntactic and functional correctness.
- The more information there is in the template file, the less information needs to be coded into the code-generation program, and the easier it is to adapt the program to generating other languages.

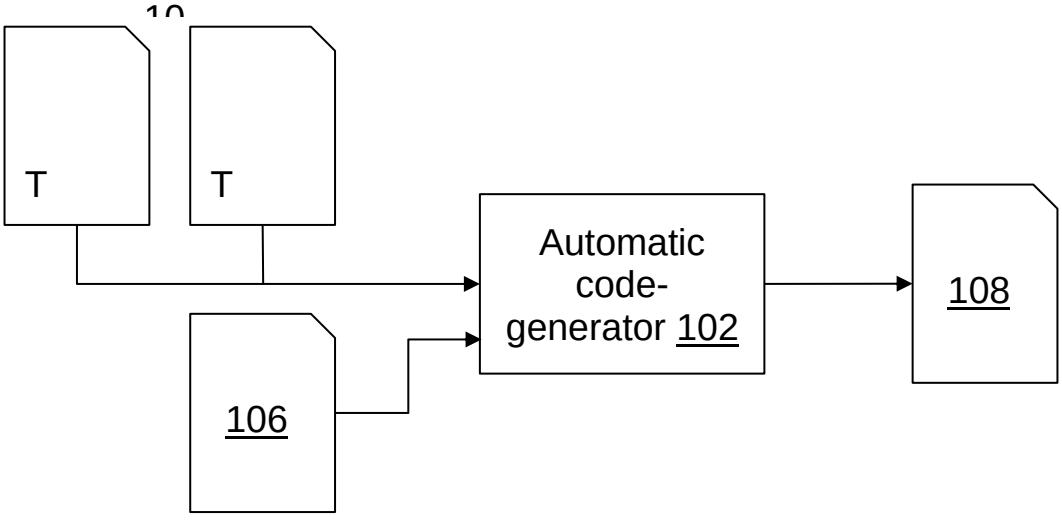

Fig. 1: Generating code with language-compliant templates

Fig. 1 illustrates generating code with language-compliant templates, per techniques of this disclosure. An automatic code generator (102) receives as input one or more pre-written

template files (104) and an input file (106) in order to generate output files (108) corresponding to the template files.

Regions of the template file are designated as snippets of text to be used in producing the output file. Snippets have names or labels. A snippet, suitably modified, can be included in the output via text substitution. The entire template file, or any portion of it, can be treated as a snippet. The template file is processed, at compile time or run time, to extract the snippets. A template file works for any valid input file.

Arbitrary strings inside snippets can be replaced by text generated by the program, including text derived from snippets, hard-coded text, programmatically transformed text, or text from the input file. A snippet inside another snippet is replaced by the name of the snippet, which can then be replaced by arbitrary generated text.

By enabling arbitrary strings to be substituted, the techniques of this disclosure give the writer of the template file the flexibility to write code that is correct and readable, has intuitive labels, and that also functions as a template. Each snippet, suitably modified, can appear in the output file as many times as necessary (including zero) to implement the objective of the input file. The ability to designate arbitrary regions of the file as extractable and re-combinable snippets confers expressive power to the automatic code generator. Nested snippets further increase the expressive power of this approach.

A typical template file may contain one instance of every pattern that the code generator may need to output. The entire file may serve as the top-level pattern.

It is useful to designate snippets inside magic comments so that snippet designations do not affect the correctness of the template file. For example, in a C++ or java input file, a magic comment can be of the following form:

```
// START snippet_name
```

In C++, a `#define` or `typedef` can be used to replace a string for compilation purposes. For example, `field_type` is not a native type in C++, but the statement

```
typedef int field_type;
```

enables templates using `field_type` as a type to compile for testing, while the string `field_type` may be replaced by any type-designating string by the generation program.

In certain implementations of this approach, there's a way to specify a namespace for snippets, such that each template file can create snippets with overlapping names. For example, two different template files can each contain snippets named `function_code` with dissimilar text. The compiler then produces text to be substituted for the string `function_code` wherever it appears in the file. The snippets may contain standard replacement-strings (which may or may not originate from nested or non-nested snippets). Some of the replacement-strings may themselves be auto-generated.

The input file to the automatic code generator (the source of the variable text) will typically be represented as a tree of nodes. Some nodes may be created by keywords, for example, `function` or `struct`. These keywords can be treated as substrings that can be combined with predefined prefixes or suffixes to generate a suitable snippet name. For example, the code generation program might prepend the string "code_for_" to the `function` or `struct` keywords. A template file for a C++ source file (.cpp) might include two snippets named `code_for_function` and `code_for_struct`, and the input tree might contain a mix of `function` and `struct` types, thus causing the output .cpp file to contain appropriate code for each function and struct listed in the input file. The corresponding template file for the C++ header file (.h) need not contain snippets with those names. As each type is encountered in traversing the tree, the prefix `code_for_` is attached to the type's suffix `function` or `struct`

to generate the name of the snippet that is appropriate for that type. Each type (node or object) in the internal tree of the automatic code generator can use and combine function invocations, hard-coded strings, and/or strings from the input and template files, to generate either or both of:

- auto-generated text, using snippet names it expects to find in the template; or
- lists of string-substitutions that can be used by higher-level tree nodes to modify their snippets.

Once all substitutions are made, the output file can be written directly from the highest-level snippet.

In certain implementations of this approach, the template file does not invoke execution of code. Instead, the automatic code generator pre-generates all replacement strings, and these are substituted into the template file as snippets. If a code generator function tries to generate text for a snippet that does not appear in the template file, an empty string results. For example, when generating a header file that does not include function bodies, and thus does not include any snippet named, e.g., `code_for_function`, the compiler quickly generates the substitution `{code_for_function: ""}` for every function declaration in the input file. That substitution is unused in generating the output file. An alternative to substituting plain strings is to have magic features inside some strings that could invoke code execution. This moves aspects of the code-generating mechanism from the code-generator into the template files at the cost of supporting what would be in effect a scripting language.

Recursion in the template file is illustrated with the following example snippet:

```
// START recursive_snippet
// ALL snippet_body
// END recursive_snippet
```

A tree structure in the compiler or automatic code generator would include nodes which calculate substitution lists that specify replacing the string `snippet_body`, wherever it occurs in the template file, with text derived from `recursive_snippet`. If one such node were the child of another, then the child node would produce an instance of `recursive_snippet` which the parent would then insert inside another instance of `recursive_snippet`. Example C++ code that implements recursive snippet output is shown below.

```
static string Snip(const string &name, const Subs &subs)
{
    SnipMap::iterator it = all_the_snippets.find(name);
    if (it == all_the_snippets.end()){
      return ;
      // This is legal, e.g. header files don't need function bodies
    }

    string snippet(it->second);
    // First, replace all the keys with uglified versions, in case
    // the key also appears in replacement text.
    for (auto & sub : subs) {
        std::regex re(sub.first);
        snippet = std::regex_replace(snippet, re, sub.first + ####);
    }

    // Then, replace the uglified keys with the replacement text.
    for (auto & sub : subs) {
        std::regex re(sub.first + ####);
        snippet = std::regex_replace(snippet, re, sub.second);
    }
    return snippet;
}
```

The above code can be used (non-recursively) by a node type `Const` which expects to find a snippet named `const` containing text `NAME` and `VAL`. (If there is no snippet `const` in the template file, then `Const::Generate()` returns without creating any new snippet text.). The code below shows a possible implementation of node type Const in the code generator.

```
string Const::Generate()
{
Subs subs {{NAME, name_->GetText()},
           {VAL, value_->GetText()}
};

return Snip("const_decl", subs); }
```

If the template file contains the following text, which is valid C++ code,

```
#define VAL 42
#define NAME (VAL) // ALL const_decl
```

then for every `Const` node in the tree, a C++ `#define` statement is produced in the corresponding output file. More complicated nodes might generate multiple candidate substitutions. For example, a node containing a list of field declarations might generate a comma-separated list of types, a comma-separated list of type-plus-name's, and a semicolon-separated list of type-plus-name's, each associated with its own text-substitution label; the template file can include whatever label is appropriate to cause the code generator to insert the correct type of list.

Alternatively, a magic (scripting language) text-substitution name might specify items such as the type of list. In one implementation of this technique, there is no restriction on strings-to-be-replaced. In another, the strings are made visually distinct in the template file, e.g., by adopting a convention of putting underscores or `ZZ` at the starts and ends of strings. Further, such prefixes/suffixes can be added by the code generation program to the strings in the substitution list right before substitution; doing so keeps its code neat, because the strings as they appear in the program do not contain the prefixes/suffixes.

CONCLUSION

Programs that auto-generate code generally produce output from a combination of fixed text and variable text. The variable text comes from an input file; the fixed text is typically

scattered throughout the program code of the code generator, which must include text for each desired format of output file. This disclosure presents techniques that instead extract the fixed text from a schema or template file written in each target language, thereby making is easier to auto-generate accurate code.